

**Fermi National Accelerator Laboratory**

**FERMILAB-TM-1934**

## **Design Notes for the Next Generation Persistent Object Manager for CAP**

**M. Isely, M. Fischler, M. Galli, A. Nigri, R. Pecanha, and K. Thayalan**

*Fermi National Accelerator Laboratory  
P.O. Box 500, Batavia, Illinois 60510*

**May 1995**



Design Notes for the Next Generation  
Persistent Object Manager for CAP

M. Isely, M. Fischler, M. Galli, A. Nigri, R. Pecanha, K. Thayalan

March 1995

\* 1 Introduction

The CAP query system software at Fermilab has several major components, including SQS (for managing the query), the retrieval system (for fetching auxiliary data), and the query software itself.

The central query software in particular is essentially a modified version of the "ptool" product created at UIC (University of Illinois at Chicago) as part of the PASS project under Bob Grossman. The original UIC version was designed for use in a single-user non-distributed Unix environment. The Fermi modifications were an attempt to permit multi-user access to a data set distributed over a set of storage nodes. (The hardware is an IBM SP-x system - a cluster of AIX POWER2 nodes with an IBM-proprietary high speed switch interconnect).

Since the implementation work of the Fermi-ized ptool, the CAP members have learned quite a bit about the nature of queries and where the current performance bottlenecks exist. This has lead us to design a persistent object manager that will overcome these problems. For backwards compatibility with ptool, the ptool persistent object API will largely be retained, but the implementation will be entirely different.

The proposed name of the new query program is [POPM], which stands for "Physics Object Persistency Manager". This name has not been settled on yet, but for the purpose of these notes, that is what we will call it. Once a name has been agreed upon, it will be changed here.

This document is a summary of the current query program, followed by a detailed collection of design notes outlining our goals and concepts behind the [POPM] implementation. It is assumed that the reader is already familiar with the persistent object API, as implemented in the UIC ptool product.

Also included in these notes are the currently understood serious limitations of the proposed system (section 7). It is strongly recommended that potential users of [POPM] (or the original ptool for that matter) carefully read that part of the design notes.

Please realize that these are a collection of notes. Even though they have been organized into a more or less document form, there are still

holes and unanswered questions about the design and implementation of [POPM]. Even in the course of writing this document, more issues have become apparent. Such problems & issues have been pointed out wherever possible in these notes, but solutions in many cases are still forthcoming.

## \* 2 Current situation

First some background.

### \*\* 2.1 UIC ptool

The original ptool product came out of the UIC PASS project, under the guidance of Bob Grossman. The persistent object API originated with this product.

The underlying implementation divided a 64 bit large persistent address space into "stores". The stores in turn were composed of "folios", each of which contained a fixed number of "segments". Folios were stored in regular Unix files, while stores were represented as a collection of these folio files, with the metadata contained in a special store "root file".

A persistent pointer therefore was composed of 64 bits, which were divided into several fixed fields (adjustable actually via C preprocessor macros). Each pointer contained:

- 1) A store number field. Each store in a persistent space was guaranteed a unique store number.
- 2) A folio number field, which identified which folio for the given store was being accessed.
- 3) A segment number field, which identified the segment within the given folio.
- 4) An offset, which identified the actual byte offset within the segment where the object was stored.

To access persistent data, individual whole segments were `mmap()`'ed into the address space of the ptool process. All of the mapped segments together composed a "slot cache" which was transparently controlled by the ptool implementation. The API defined a persistent pointer "type" (a class actually) which overloaded the C++ dereference operator ("`->`") such that use of it resulted in a look-up into the slot cache. If the segment containing the persistent object was present, a "real" pointer to the offset of that object within the segment was returned. If the segment was not present, another segment of the slot cache was remapped to the correct segment of the correct folio in the correct store, and a "real" pointer to the object offset inside that segment was returned.

Since persistent data was always `mmap()`'ed, sparse access was handled efficiently (i.e. only parts of the segment being touched would actually be faulted in), and modified data write-back was also automatic (the OS will recognize dirty pages and transparently perform the needed write-back into the folio file).

### \*\* 2.2 Fermi-ized ptool

The Fermilab CAP effort viewed ptool as a possible way to efficiently

organize the immense amounts of data that experiments need to have online for their data mining requirements. However the system we had to work with was an IBM SP-2 system. It is a machine composed of RS/6000 CPUs interconnected by a high-speed message passing switch. The disk storage is physically distributed across the I/O nodes. There is no "single system image", in that each node looks unique, and disks are only visible on the local node where the particular disk(s) were attached.

We looked at the UIC ptool product and sized up our situation on the SP-2. The following problems were found:

- 1) The ptool implementation used mmap, which will only work with locally visible file systems. The I/O disk file systems were not visible outside of the I/O nodes (unless NFS was used, but this caused other problems).
- 2) Processes using the switch must all be started at one time, and must finish as a unit. There was no capability to model a system where a set of servers would always be running on the I/O nodes and query processes would be able to come and go according to the demands of the moment.
- 3) The API for accessing the switch in its "fast" mode only permitted one process per node; multiple processes on a single node were not able to implicitly share switch access. TCP/IP over the switch was available as an alternative (which then conceivably could have allowed solving issue #1 with NFS), but this lowered the available maximum bandwidth to about 8MB/sec (going directly permitted up to 30MB/sec). We determined that penalty to be too high. But the "fast" mode was also problematic because we really needed to be able to run multiple queries on a single node.

We finessed issue #1 by eliminating mmap() and doing explicit reads into the former slot cache data slots. Writes were not permitted, meaning that the distributed system was not able to modify any stores. Instead the stores / folios were created using the conventional ptool with mmap() running on a local node; then the store files were explicitly spread out to the I/O nodes via a standard Unix rcp command.

Issues #2 and #3 were worked around by implementing a set of static servers that ran 100% of the time on all nodes. On the compute nodes, the servers created SYSV IPC message queues & shared memory through which query processes would request segments containing persistent objects; in that fashion queries could come and go just by using the message queues & mapping the shared memory. When segments were fetched, the local server would copy the segment into the shared memory, and the requesting ptool process would then use point to that as the slot cache data. On the I/O nodes, I/O servers listened for messages from the switch and then performed the operation, forwarding the results back to the compute node.

The solution on the surface appears sound, but it had many problems. The biggest one recognized first was that when a single query was taking place, there was no parallelization of I/O. A query running in this fashion would pound on the I/O nodes in sequence as it scanned through the folios of a given store. The net result was that I/O throughput could never surpass the rate of one disk drive.

To overcome this bottleneck, a read-ahead strategy was implemented on the I/O nodes. Whenever a ptool process started sequencing through a store, all the I/O nodes attempted to analyze the access pattern. If it was

sequential, then the I/O nodes would attempt to read-ahead so as to effectively parallelize the disk access. The read-ahead depth would depend upon how many read-ahead streams were running and how much memory was available on the I/O nodes. Multiple disk reader processes were implemented to make overlapped I/O possible on the I/O nodes.

This worked to some degree and the overall I/O throughput increased. However it still had problems. The biggest problem was that since segments were assigned sequentially through the folios and that folios (at this time) were 4MB, then at least 4MB times the number of disks per I/O node times the number of I/O nodes was required. For example, with 4 I/O nodes and 4 disks per I/O node, then 64MB of read-ahead depth over all the I/O nodes was required for maximal efficiency. The I/O nodes only had 64MB of physical memory each, and only about half was available for disk buffering! Most stores were not even big enough to benefit from this. For example, if a store was only 4MB in size, then only one folio could be used and there would be no chance for efficient read-ahead.

This system suffered from other major problems as well. To list a few:

- 1) When a server on a compute node is fetching a segment, it is completely blocked. No other query processes on that node will be able to fetch in parallel. This effect severely restricted the potential utilization of the switch.
- 2) When a server on an I/O node is performing a fetch, no other compute node servers would be able to talk to it. In other words, an I/O node server is completely locked to the requesting compute node server when a transaction is taking place. This further restricted switch utilization, and created a bottleneck on each I/O node. A better approach would have been to look for other requests during the time that the disk is being read for the current request.
- 3) The basic striping unit in this system is the folio. This meant that to increase striping efficiency, the folio size had to shrink. If the segments had been sequenced across the folios instead of within them, then the effective striping unit size would have been the segment. This would have enabled far better read-ahead behavior because the read-ahead depth per I/O node would only need to be a few segment's worth, instead of whole folios. In addition, small stores would then still get the maximum read-ahead benefit.
- 4) The read-ahead buffer is on the I/O nodes. Assuming problems #1 and #2 didn't exist, then placing the read-ahead buffering in the compute nodes would have opened up opportunities for more overlapped I/O over the switch, which would have allowed us to better utilize the switch thereby raising the effective bandwidth. This technique would also have permitted us to bury all of the intervening latency - switch latency, I/O node process context switch latency, and several memory-to-memory copies worth of latency. Finally, placing the read-ahead detection logic in the compute nodes will significantly simplify the implementation.
- 5) Several memory-to-memory copies were taking place that could have been avoided.

## **\*\* 2.3 Summary**

That's the current situation. These design notes suggest a new design that should eliminate nearly all of the above bottlenecks, lift many of

the inherent limits (of both UIC ptool and the Fermi specific limits), and at the same time provide for a much cleaner implementation.

The following sections detail the new concepts, implementation ideas for these concepts, and a rough analysis of how this will significantly enhance overall performance of the system.

### \* 3 New concepts

Listed below are the new concepts and ideas relative to the original UIC ptool product. Implementation ideas relative to the concepts are also provided where appropriate.

#### \*\* 3.1 API enhancements

It is the goal of [POPM] that it should be possible to run programs originally written for ptool with little, perhaps zero, modifications. However it is felt that some aspects of the persistent object API impose unnecessary limitations on the system. Explained below are [POPM] concepts that enhance the original persistent object API with the goal of improving both flexibility and performance of the system.

Note that these concepts are additions to, not modifications of, the original API.

##### \*\*\* 3.1.1 Flexible persistent pointer bit fields

The UIC ptool product hard-codes the bit field allocations (i.e. maximum folios per store, maximum segments per folio) in a 64 bit persistent pointer at compile time. Unfortunately, that doesn't provide enough space in some cases. [POPM] will implement a more flexible scheme, where the store number size is encoded into the upper two bits of the persistent pointer (like IP address classes), and the remaining bit field sizes are defined in the store's metadata when the store is created. This will permit pushing any edge of this address space "envelope" at run time, on a store-by-store basis. Note that with the upper 2 pptr bits set to zero, and appropriate parameters for the other sizes, compatibility can be maintained with store data created by the existing UIC ptool.

In addition, the lower 32 bits are no longer reserved for just the offset portion of a persistent pointer. It is really unlikely that segments could ever be 4GB wide (in fact, the segment size will be fixed by [POPM]). Folio number and segment number fields now also may occupy part of the lower 32 bits, significantly increasing flexibility even further.

Previously, the lower 4 bytes mapped an offset within a segment, while the upper 4 bytes specified store number, folio number, and segment number. Now, the store number field size is determined by the upper two bits, the offset field size is fixed to 16 bits (assuming 64KB segments; see section 3.3.2 for details), and the remaining bits are available for allocation as any combination of folio number and segment number bits. Here is a map of a 64 bit persistent pointer, showing the three cases of how the store number is mapped:

```
01sssssss ssssssss ssssxxxx xxxxxxxx xxxxxxxx xxxxxxxx 00000000 00000000
00sssssss ssssssss xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx 00000000 00000000
1sssssss xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx 00000000 00000000
```

The lower 2 bytes ("o") map an offset within a segment. The remaining 6

bytes are divided into a store number part ("s") and a folio / segment number part ("x"); the upper 2 bits control that division. The exact division between folio & segment number is programmed on a per store basis (like the segment size). Note that like TCP/IP IP numbers, the size of the upper field is determined by the top two bits, thus maximizing flexibility.

Variation on the theme:

About the only thing in the above definition that is hardcoded is allocation for the store number field. There is a way to make that totally flexible however. The answer is to maintain some address-space specific metadata (see section 3.1.3 on multiple address spaces) that can be used to find the store number field. When a [POPM] application sets its address space, this metadata can be read. With this ability, the store number field can be made completely arbitrary.

This last bit is just a thought for now; the metadata required would have to be stored in the address space-specific dbmap file, and there is some issues remaining about how that file will be formatted & accessed. For now it's best to keep this idea out of the dbmap file and employ the 3 class approach. But we can change to this other store number approach fairly easily down the road, once the dbmap issues are understood.

### \*\*\* 3.1.2 Physical pointers

Though the persistent pointer dereference overhead isn't too much, it is still too heavy when running codes that populate object stores. In this case, every field of a given object is going to be dereferenced (assuming the use of a null constructor - see section 7.2.2 for the explanation), resulting in a very large number of redundant executions of the dereference operation.

[POPM] implements a new element to the API called a "physical pointer". It can effectively hold onto a pre-dereferenced persistent pointer. Accessing data through a physical pointer is very fast; in fact the access function is merely a single type cast. With this new API concept, population code will only take a hit of 1 actual dereference per object instead of 1 dereference per object member per object.

This also addresses a problem with persistent pointer lifetimes, a subject discussed in section 4.4.4.

### \*\*\* 3.1.3 Multiple address spaces

In the UIC persistent object design, all object stores effectively reside in a single large 64 bit address space. Because of the single address space, there is little isolation between stores. For reliability, policy, and practical reasons there must be an ability to support multiple address spaces. [POPM] will explicitly provide this.

Before any store can be opened in a [POPM] application, the address space must be selected - either via some kind of global function in the program, or perhaps through an environment variable (some kind of permission checking mechanism may also be employed here). Once the address space has been picked, stores may be opened and accessed in the usual way. Though it is not possible for a single [POPM] application to have concurrent access to multiple address spaces, a running query system can have different [POPM] processes running, each using different address spaces.

Multiple address spaces will allow several different data models to be imported onto the system at once, and will also allow different experiments to coexist on the same system.

### \*\*\* 3.1.4 Persistent pointers as templates

Instead of a macro (PPTR) to define a persistent pointer to a given object type, [POPM] will use a more natural template instead. Presumably the original authors of UIC ptool had intended for this to be a template, but compiler technology at the time wasn't quite up to the task.

The template will be designed to eliminate code bloat problems usually associated with template classes. How? By using a non-template base class which contains all of the persistent member functions. The template class will just be short inline code containing only a few type casts. It should compile down to virtually nothing.

## \*\* 3.2 Distributed implementation

The UIC ptool product provides local mapping to store data for a single process at a time. Portions of local files are mmap()'ed into the process address space as the program runs. This is a fine approach, but it is not appropriate if the desire is to raise I/O throughput beyond what a few disks on a single host can provide.

One of the goals of [POPM] is to provide an inherent ability to distribute and stripe store data across multiple hosts in a cluster. Assuming a fast (i.e. > 10MB/s) interconnect, it should be possible to have an effective I/O throughput for a single query that far exceeds what is possible on a single node. In addition, the distributed approach should also permit efficient scaling up and sharing of I/O throughput as more [POPM] applications run.

Sections described here below explain the new concepts that should make this fast distributed I/O possible.

### \*\*\* 3.2.1 Explicit I/O with dirty flags

The UIC ptool implementation employs an internal "slot cache" containing the last few segments that were mapped in response to the dereferencing of a persistent pointer. Each slot cache element is in fact an mmap()'ed section of a folio file.

In a distributed implementation, use of mmap() is no longer possible because the files being accessed are no longer directly accessible to the local node. The [POPM] implementation will instead perform direct reads & writes to existing memory in the slot cache.

There are two advantages to using mmap() over the direct reading and writing of file data:

- (1) Since the mmap()'ed page can be demand-faulted in, extraneous disk activity can be avoided if only a small part of the segment is ever touched.
- (2) Writes are automatically optimized because the virtual memory subsystem of the Unix kernel will only write-back pages which have really changed by the system when the segment is unmapped.

The [POPM] implementation does not try to solve issue (1); it is believed that there would be nothing to gain anyway because a) reads over the SP-2 system's switch are optimal when large blocks are read, and b) It is likely that most of the segment will probably be touched anyway by the application.

Issue (2) is a different story. The [POPM] implementation will in fact be able to efficiently determine when a segment needs a write-back at appropriate times by: a) modifying the slot mapping protections such that it is read-only, b) catching a SIGSEGV if an application attempts modification, and c) setting a "dirty flag" & re-enabling read/write access for the slot in the SEGV handler if the faulting address was within a protected slot AND the store was opened in a writable mode (and then of course allowing the handler to return and the application to continue). The dirty flag can then be used to execute a write-back when the slot needs to be re-cycled. Note that it will also be possible for the application to explicitly force the dirty flag in certain cases when it is known that a write-back will be needed (i.e. for newly created segments). Thus the signalling and protection changing overhead can even be avoided in these situations.

With the ability to write-back segments in this fashion, then it will be possible to populate distributed object stores far more efficiently than is currently being performed.

### \*\*\* 3.2.2 Slot cache in shared memory

[POPM] employs a slot cache, but it is no longer fixed in size, and is instead placed into a large block of shared memory. The size would be considerably larger, probably 128 or 256 entries @ 64KB/slot. All [POPM] applications running on a single node share the same slot cache.

This approach has two notable benefits:

- (1) It now becomes possible to dynamically partition the number of slot cache entries to each process according to demand. For example, if only one [POPM] process were running, it would be able to use all of the slots of this much larger cache. As more [POPM] processes run on a node, the cache would be divided among them. If a particular process happens to have a larger "working set" (if this concept is even useful; not sure yet), then it may get proportionally more slot cache entries.
- (2) A memory-to-memory copy is avoided. This is important since the slot I/O is performed with explicit reads & writes instead of direct file mapping. Note that the actual reads and writes will be performed by external server processes (see the implementation section for an explanation). Since the slot cache data is externally visible to other processes, there is no requirement for an extra copy operation into (or out of) a [POPM] process's internal data segment when a slot is read (or written).

### \*\*\* 3.2.3 Segment read-ahead

Read-ahead is a concept where the underlying system attempts to anticipate the needs of the application and brings in the next few segments before they are actually asked for, thus eliminating any wait when the segment data is finally needed. Enabling multiple segment reads to run concurrently with the operation of the application therefore reduces process blockage due to segment waits, and also opens up

opportunities for parallelizing and overlapping I/O into the system. Read-ahead results in some very significant benefits, so it pays to provide it.

The UIC ptool product does not provide any kind of explicit read-ahead. The current Fermilab-modified multi-node version of the UIC ptool implementation does provide it, but it is not correctly implemented. The rest of this section compares [POPM] read-ahead to the current Fermi implementation of read-ahead:

The Fermi-modified ptool does its read-ahead at the level of the I/O nodes, which has several drawbacks:

- 1) Detection of read-ahead is performed in the I/O nodes by analyzing the pattern of access from a given client. Presumably, the choice to do it in the I/O node was on the belief that the right place for computing the amount of read-ahead should be where it is possible to balance the amount of available memory in the I/O node. However the detection is more difficult here, and the memory size of the I/O node becomes irrelevant when the actual read-ahead cache is kept in the compute nodes.
- 2) All read-ahead parallelization & caching is only happening in the I/O nodes. Unfortunately, this scheme does not allow for any overlapped parallel I/O outside of the actual I/O nodes (i.e. in the switch path or on the compute nodes).

[POPM] will implement read-ahead in the application process itself, where it is easier to detect, and where it will do the most good. When a user application starts sequencing through a list, read-ahead can be performed. When this happens, [POPM] sets up and requests reads for multiple segments, instead of the one being blocked on (but continues as soon as the blocked-on segment arrives). The amount of read-ahead is again a function of available memory, but this time it is the slot memory in the compute node, not the server. And since the memory at issue is simply the slot cache, this is made easy. Remember that a single slot cache is now being shared by all [POPM] processes on a node.

Another problem the Fermi implementation has is that no parallelization can occur at all if the read-ahead distance does not exceed the size of one folio. This is because when reading ahead only a few segments, all of the reads are going to be within a single folio file, since there is no striping of segments across folios. In order to cause parallel accesses, therefore the read-ahead distance must be large (4MB/folio in typical use, meaning more than 64 segments of read-ahead is required). [POPM] will implement segment striping (see section 3.3.1). This will permit much better read-ahead performance with smaller read-ahead distances (as little as 2 segments worth of read-ahead can be distributed over multiple I/O nodes).

### \*\*\* 3.2.5 Storage units & folio striping

First, a sidebar about "storage units":

A distributed I/O system is composed of one or more I/O nodes. Each I/O node is a CPU connected to one or more "storage units". The storage unit is the defined unit of I/O striping, namely that I/O operations pending to different storage units should in theory be able to proceed in parallel. I/O operations pending to the same storage unit will always be scheduled sequentially, one after another.

Typically, a storage unit is a disk drive.

So, given the above, then for example a distributed I/O system composed of 8 I/O nodes with 4 disks on each node actually contains 32 "storage units".

With that concept understood, examine now the issue of distributing persistent store data across an I/O system. The UIC ptool implementation of course was not designed with this in mind since it doesn't allow distributed data, but [POPM] is intended for exactly this use.

In [POPM], distribution of the data means spreading the folios evenly across the storage units. So what is the pattern to the scattering? There are two cases that determine how to locate the folios: accessing existing folios, and creating new ones.

When accessing existing folios, they are allowed to exist on arbitrary storage units; the [POPM] implementation will always be able to locate the proper storage unit when a given folio is required. This means of course that all folio file names must remain unique across all candidate storage units.

When creating new folios, the initial location is determined via an algorithm, whose parameters are set when the store in which the folio is a member is created. The parameters are as follows:

- a) The set of storage units over which store data is distributed. Logical storage unit numbers are assigned to each storage unit in sequence according to the order of declaration. For example, if the following storage units (using semi-arbitrary naming conventions) "capfcn1:/spool0", "capfcn1:/spool1", "capfcn3:/spool1", "capfcn5:/spool2" were declared when creating store "foobar", then the following association is created for numbering the logical units:

```
capfcn1:/spool0 == unit0
capfcn1:/spool1 == unit1
capfcn3:/spool1 == unit2
capfcn5:/spool2 == unit3
```

- b) The horizontal striping factor. This determines how many "striping groups" there are. The horizontal striping factor is the number of logical storage units to group together per striping group. For example, a horizontal striping factor of 2 in a set of 8 logical storage units results 4 striping groups, each containing 2 storage units apiece. NOTE: The number of logical storage units declared for the store must be an even multiple of the horizontal striping factor.
- c) The vertical striping factor. This determines how many folios are assigned to each storage unit in a striping group. For example, with a vertical striping factor of 2, each storage unit will get 2 folios apiece.

Assignment is made in order of horizontal striping factor first, then by vertical striping factor (this is unfortunately different than the segment striping since the limit is the number of storage units, not the size of a storage unit - see section 3.3.1). If all of the striping groups fill up, then assignment starts again back at the first striping group. Here's an example numbering scheme:

Horizontal striping factor: 4  
 Vertical striping factor: 3  
 Number of storage units: 8

unit0	unit1	unit2	unit3	unit4	unit5	unit6	unit7
0	1	2	3	12	13	14	15
4	5	6	7	16	17	18	19
8	9	10	11	20	21	22	23
24	25	26	27	.	.	.	.
28	29	.	.				
.	.	.	.				

Again it is important to note that this rigid organization is only employed when creating the folios; once they are created they are allowed to exist in any organization. Folios may be freely shifted around and the system will still be able to function. The reason for this is that if a storage unit fails, the folios it contained can be restored onto other storage units without impacting the overall data integrity.

The information describing the folio striping organization is stored as part of the store's metadata.

## \*\* 3.3 Other performance enhancements / differences

### \*\*\* 3.3.1 Segment striping

In the UIC ptool implementation, segments are numbered within folios first before going to the next folio. Contrasted with that, [POPM] stripes segments across groups of folios. For example, segments can be striped within groups of four folios at a time. This should permit much finer grained scattering of the data across disks and open up much better opportunities for the read-ahead logic.

The horizontal striping "factor" for a given store defines the number of folios that make up a striping group. Striping groups are composed starting from the first folio in the store. Segments are striped across this group until the folios are filled, at which point the next striping group is used.

For example, with a horizontal striping factor of 4, the segments would be ordered like (where n= number of segment bits in the pptr):

folio0	folio1	folio2	folio3	folio4	folio5	folio6
0	1	2	3	$2^{n*4+0}$	$2^{n*4+1}$	$2^{n*4+2}$
4	5	6	7	$2^{n*4+4}$	$2^{n*4+5}$	$2^{n*4+6}$
8	9	10	11	.	.	.
.	.	.	.			

In addition to horizontal striping, [POPM] will allow for a vertical striping factor associated with the store metadata. This defines the number of consecutive segments to assign to a folio before moving to the next horizontal stripe. The above example therefore assumed a vertical striping factor of 1. With a vertical striping factor of 2 however, it would change to:

folio0	folio1	folio2	folio3
0	2	4	6
1	3	5	7
8	10	12	14

It is not certain whether the vertical striping factor will allow for any kind of performance improvement; what it does is to effectively increase the chunk size of the stripe.

Important note: Though the terms are similar, this striping information is not logically the same as that used for the folio striping as described in section 3.2.5. Why? Because folios have a fixed size, which is different than the "storage units" described as part of the folio striping, which logically don't have a size limit. Also note that though there is no (effective) limit on the number of folios, the number of storage units by comparison is small and fixed up front. In other words, the segment striping algorithm has a fixed overall vertical dimension, but an arbitrary horizontal dimension. The folio striping algorithm however has an arbitrary vertical dimension but a fixed horizontal dimension. Therefore the striping description has to be different.

Note that a horizontal striping factor of 1 (with any arbitrary vertical striping factor) would result in a segment layout that matches the original UIC ptool.

### \*\*\* 3.3.2 Non-adjustable segment size

Per-store adjustment of the segment size is not permitted in [POPM]. All segments will be a fixed size (the exact value is not quite decided yet, either 64K or 128K). Why do this? It significantly simplifies the underlying implementation. This point is very important; the entire implementation described in section 4 of these notes hinges on a large central fixed array of slots. By its very nature, this array has a characteristic "slot" size, which is the segment size. Slots are dynamically swapped among stores and query processes as the demand warrants (see section 4). To permit a per-store segment size would severely complicate this slot sharing behavior, as now slots would have to change size as they are swapped around.

A possible way to do this might be to divide storage for large segments into smaller ones (along power of 2 boundaries) as required by store's segment size. However this may lead to fragmentation problems - a large segment may be needed when there aren't enough continuous fragments to assemble one. In addition, this doesn't even begin to address the allocation policy of such a system, nor the problem of guaranteeing enough segment header control "elements" (see section 4.1.1). Another possible way to accomplish this might be to just declare an very large slot size and just not use all of it for smaller segments. However this just wastes space, and still imposes an upper limit on the segment size. The price for allowing variable length segments is just too high.

Given a fixed segment size, then what size is appropriate? The size must be reasonably large in order to permit efficient distributed I/O, but not so large that useless data is always being read in. Also keep in mind that since the smallest unit of striping is the segment, that if segments are too large then striping will also suffer.

There is a hidden problem here: It is impossible to permit objects to cross segment boundaries. This is a limitation inherent in the persistent object API. Therefore it is a limitation in the UIC ptool implementation, and will also be a limit with [POPM]. The problem is

that the compiler hides the information about the offset within the object that is being dereferenced, meaning that if an object crosses a segment boundary, the implementation is unable to tell which side of the boundary the particular dereference really needs. Therefore segments must be at least as large as the largest object in the system. There is no way around this without significantly altering / damaging the persistent object API.

The segment size will (for what it's worth) be a compile-time constant for [POPM].

### \*\*\* 3.3.3 Store migration

[Note: This concept is very new; at the current time very little work is being done to address it.]

The underlying storage format of data as defined by the persistent object API, is not very portable. Three major factors affect the format:

- 1) Folio size. The correct choice for folio size depends largely on the medium of storage. For example, disk-based folios are best kept relatively small (i.e. 5 to 20 MB), but folios coming off of a robotic-tape backed HSM should probably be much larger (i.e. 100 to 200MB). An incorrectly chosen folio size will not render the object data unreadable, but performance may be suboptimal: disk based folios that are too large may not be stripable because the entire store will fit into a single folio, and tape based folios that are too small may result in excessive tape seek overhead.
- 2) Segment size. The correct choice for segment size depends on the communications medium between storage units and the compute nodes. If the size is too large, communications & disk bandwidth will be wasted because data will be fetched that is never used. If the size is too small, communications overhead & disk latency will dominate and bandwidth will suffer again. One other implementation aspect of this: since objects may not cross segment boundaries, segments must at least be large enough to hold one complete instance of a single object, meaning that object sizes are also limited by the segment size. Because of these problems, differing segment sizes may be employed on different platforms. Since the segment size is fixed in [POPM], reading other object data sets with different segment sizes will require a recompile, making portability a problem.
- 3) Architecture & compiler dependencies. The CPU byte order can be different, and different compilers may pad object structures in completely different ways.

Factor (1) makes portability merely uncomfortable. Factor (2) is worse than uncomfortable. Factor (3) renders object data portability impossible, in the "native" format.

The native UIC ptool product does not address these issues. [POPM] will. Here's how. There will be a method / procedure / function in [POPM] for migrating stores, importing stores, and exporting stores.

For store migration, one needs only to rewrite the store in the new organization. The output format is still the same (i.e. native), but the organization can shift, which means it will be possible adjust the store's metadata (i.e. striping factors & folio size).

An example of store migration would include situations where a store currently in object format on an HSM (large folio sizes, no striping) would be moved to a distributed disk system (small folio sizes, extensive striping). Another example might include re-striping a store across additional / fewer storage units (i.e. changing the segment striping).

For store import / export, an entirely new file format is needed. This format would have to be stream-based, with no padding, and embedded metadata describing the structures. The format would not have to be "efficient" in terms of accessibility, but it should be efficient with respect to compactness. The idea here is that an import / export program could be designed that would read & write this new format. Then to move store data from one platform to another, export it to this stream format at the source, and import the stream back into native object format at the destination.

This approach will solve all but one problem: too small of segment size. If the source object store has any objects that are larger than the segment size of the destination, the operation will fail. The only way to solve this problem is to either (a) make it possible to easily vary the segment size (not easy - potential big efficiency hit), or (b) make it possible for objects to cross segment boundaries (impossible to do without changing the API - the compiler hides the object offset being read when a persistent pointer is being dereferenced).

There is no proposed implementation yet of the import / export process. There are a few unsolved problem. For starters, the object data structure must be known to [POPM] in order to import or export them. This information is lost once the object source code is compiled. A possible solution would be to require import / export member functions for the persistent objects. This changes the API though.

The migration process can be performed without direct knowledge of the object data structures.

In the current system at Fermi, the import / export function is effectively being done now by using Zebra format as the import / export format. This idea does not of course eliminate the Zebra format (since the original source data is Zebra format), however implementing it will make [POPM] more generally useful.

#### \* 4 Implementation

The following sections attempt to put the above concepts into perspective, and show how a large query system can be efficiently implemented, specifically on the SP-2 hardware.

#### \*\* 4.1 The big picture from a single node perspective

For the sake of simplicity, a single node system is first assumed. With a single host, the message passing layer and I/O servers processes are not relevant. After the single node configuration is discussed, the explanation will be expanded to show how it works in the multiple node configuration.

The following elements exist in a single node query system:

Shared slot cache - Repository for all slot data, slot control information, process control information, and node-global control information. Only one instance of this exists, and it is mapped by

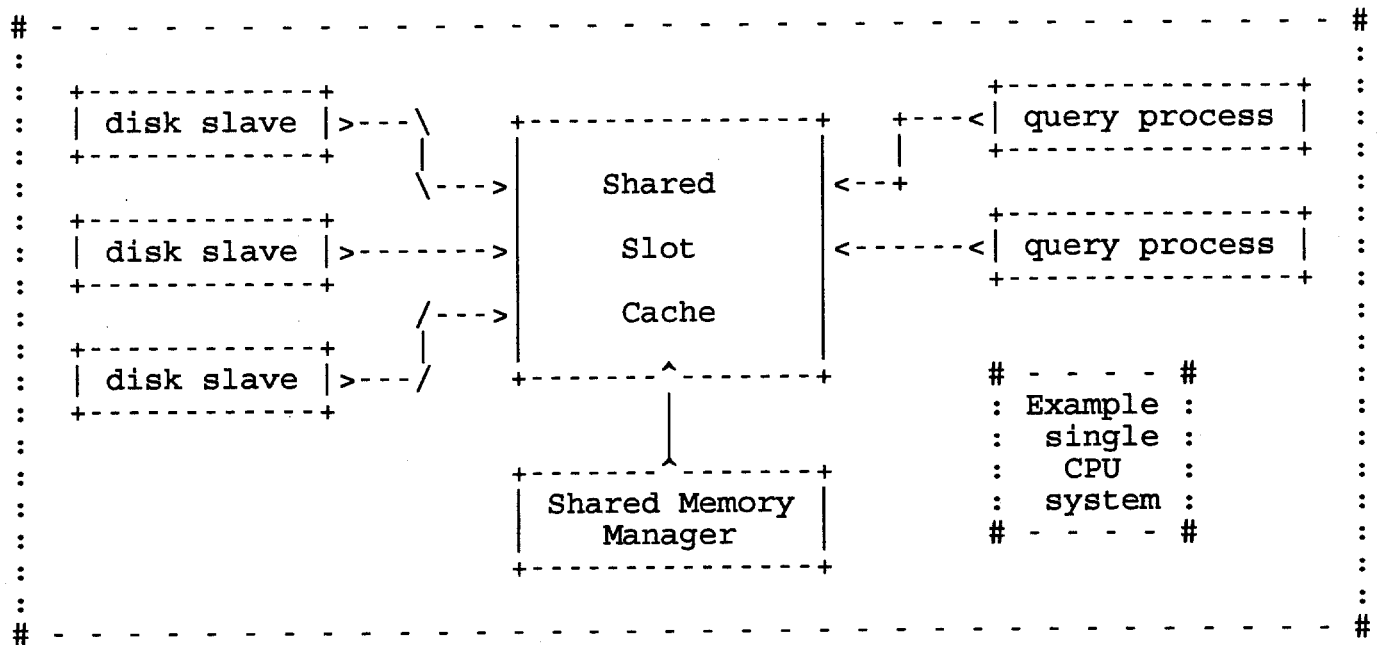
all processes in the system. (This was introduced in section 3.2.2.)

Disk slave - Performs the actual segment fetching / writeback with the physical system. The disk slave functions by locating a slot with a pending I/O status and performs the requested operation. There may be any number of disk slaves running in parallel on a node (presumably a good number that maximizes overlapped I/O is 2 disk slaves per storage unit). With multiple disk slaves, multiple slot reads / write may be taking place at once. The disk slave is always doing one of two things: Either it is looking for work or it is performing I/O.

Query process - This is the analog to the current ptool program instance. Each running query process represents one running query. Segment data is accessed in the shared slot cache.

Shared memory manager - This is the process responsible for maintaining overall integrity of the system (for this node). It has two main jobs: (1) Managing allocation of slots, and (2) cleaning up after query processes when they exit.

Here is how the pieces fit together for a single node:



(Fig 1 - single CPU system)

Here's how operation normally proceeds:

1. A query process dereferences a persistent pointer. This results in a hashed search of its allocated slots. If the slot is present, a pointer to it is generated and the operation is complete.
2. If the slot is not present, the query process allocates a new slot (from a free list), and fills in the control information indicating what segment of what file is required.
3. At this point, a disk slave sees the request and performs the requested I/O.
4. When the I/O operation is complete, the query process generates a

pointer to the new data and the operation is complete.

There are of course many other important details that make this concept efficient, but we'll get to that in the relevant sections further on.

Following are sections describing the above parts in detail.

#### \*\*\* 4.1.1 Shared memory

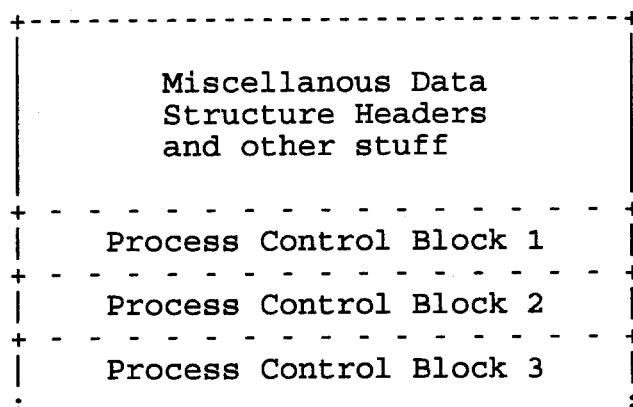
The shared memory contains more than just slots. It is actually composed of 3 major sections:

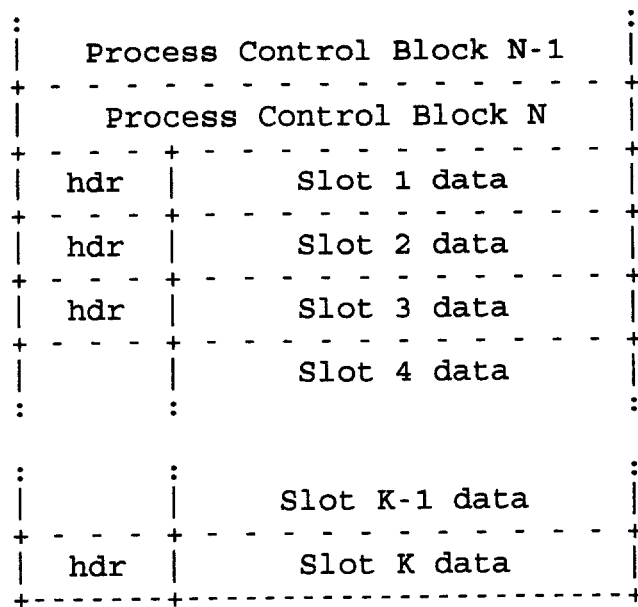
**Slots** - There is a fixed size array of slots (expected to be around 128). The array is actually two parallel arrays: One for the actual slot data, and one is a "header" for the control information. The slot data in reality will be contained in a separate block of shared memory (for page alignment and other implementation reasons), however logically it is part of the same overall block. The slot control information consists of fields describing the slot contents, fields describing any pending operation, and various linked list pointers for use by other data structures.

**Process control blocks** - This is a fixed size array, where each element represents a process that is using the shared memory. Whenever any process maps the shared memory (be it a disk slave, shared memory manager, query process, etc), it must allocate an element of this array and fill in the requisite information. This enables certain forms of locking that are required, permits per-process linked-list data structures to be assigned, and gives the shared memory manager a means to learn about all processes in the system. The size of this array limits the number of processes that may connect (but the element size is small so we can make this very large without significant penalty).

**Global control information** - This is a single data structure containing various fields for controlling the overall system. Examples of some of the global information include a linked list of free slots, a linked list of slots pending for reads, a linked list of slots pending for writes, a linked list of disk slaves waiting for work, counts of the number of processes in the system, and the key value for the slot data shared memory block.

Just to further cement this description, here is an illustration of the logical layout of the shared memory slot cache (with N process control blocks and K slots):





(Fig 2 - shared slot cache layout)

### \*\*\* 4.1.2 Disk slaves

As said before, disk slave processes are responsible for actually shipping segment data to / from persistent storage. Logically, a given disk slave process is always looking for a slot with a pending I/O operation. Upon finding one, it performs the request, leaving the results (completion code plus data if a read) in the slot. Disk slaves are always looking for I/O to perform. Since there is no real limit on the number of disk slaves, and since a single operation can be completely described in a slot and performed by a single disk slave, then the maximum number of parallel I/O operations is limited only by the number of disk slaves or the number of slots (whichever is less).

The actual implementation is more efficient than this description suggests. In reality, all pending I/O operations are collected in a few linked lists (pending reads, pending writes, and pending reads with priority) so no searching is required. In addition, when a disk slave fails to find any work to do, it will put its process control block onto a linked lists of disk slaves waiting for work and then wait for a signal. When a new I/O request becomes pending, the first PCB on that list will be pulled and signalled. So there actually won't be any busy-waits either.

It is interesting to note here that no assumption has been made about what kind of storage actually exists at the other side of the disk slave. Certainly, a Unix file system may be trivially employed. But a raw disk partition could also be used (with the disk slaves imposing their own file system) - this might be considerably faster when one considers that since the slot cache is effectively the cache, then the Unix buffer cache will probably be wasted. An HSM could also be employed as the disk slave back end...

### \*\*\* 4.1.3 The [POPM] client library (query process)

The [POPM] client library, when linked with query code, becomes the query process. It is at this level where the persistent object API exists, and it is at this level where all the persistent object specific concepts are implemented.

All persistent pointer dereferences performed in the query are translated to segment addresses. In concept, this address is used to locate a slot in the shared cache, and pointer to it is used by the query code for accessing the persistent data.

In practice, much more than that will be happening. The following important algorithms / concepts are also a part the query process:

Hashed segment search - In order to make persistent pointer dereferencing as fast as possible, the 64 bit pointer is first changed to a tag (by masking off the offset portion) and that tag is hashed to a chain contained in the query process's PCB (Process Control Block in the shared slot cache) which is then linearly searched for a previously set slot tag matching the computed tag. This operation needs to be extremely fast, hence the tags and hashing. Hopefully with a good hashing function and enough hash chains, the linear search should only average one or two lookups.

Slot allocation - Shared slots are in reality private to a given query process instance (but they are in shared memory to allow for dynamic balancing and to eliminate extraneous copying when fetched / returned). A list of the slots that a process "owns" is kept in the PCB for the process. Query processes must therefore allocate empty slots from a free list of slots kept in the global section of the shared memory. If that list is empty, the query process must wait for more slots to become available, by placing itself on a resource-wait list, signalling the shared memory manager, and then blocking until a signal is received.

Slot stealing - The shared memory manager is able to steal slots from query processes. It does this in order to keep the free list at an optimal level. This behavior means that it is possible for slots to "go away behind the back" of the query process. To guarantee correct behavior under this circumstance, a locking mechanism must be employed. The "physical pointer" concept described in section 3.1.2 is a natural consequence of this. But there's a more sinister problem lurking - if an "old style" persistent pointer is directly dereferenced, there is a period of time when the query process has a direct pointer into the slot but no lock to protect it. Remember that the API provided by the UIC ptool implementation did not provide for the concept of a pointer dereference lifetime. If the timing were really bad, the shared memory manager could yank back the slot before the query process has a chance to read the required data from it. To solve this problem, a single global lock (actually an instance of the base class for the physical pointer) is employed. This lock is always used for the last persistent pointer dereferenced, in other words the last slot pointed to always remains locked. When another dereference takes place, the global lock is released from that last slot and reset to the newly dereferenced slot. This guarantees validity for persistent pointer dereferences up until the next persistent dereference, and has an effect similar to the UIC ptool implementation - actually we can mimic the UIC ptool pointer lifetimes by using an array (of size equal to the old slot cache) of global locks with a circulating pointer to the next to reset.

Store root file access - When a store is opened, a "root" file must be opened by the query process so that the store's metadata may be retrieved (or updated). The UIC ptool code directly opened and

manipulated root files. [POPM] will do something different. The same I/O path used to reach the folios will be used to access the root files. Specifically, a root file will be mapped into a slot just like another segment, using the same mechanisms available for segment I/O. This will permit the same uniform access for root files as that available for folios (which becomes much more important in the multiple node configuration). Note that this works because below the level of the query process, everything is just seen as blocks of arbitrary files - there is no notion of stores, folio, segments or anything ptool-ish below the actual code for the [POPM] client library in the query process.

Dbmap file access - The "dbmap" file maps store numbers to stores. This is required for allocating unique store numbers when creating new stores in an address space. It is also useful for locating the proper store when an arbitrary persistent pointer is dereferenced (but this is not expected to be important for datamining applications envisioned). There must be one and ONLY ONE unique dbmap file per address space. Again, this file will be mappable in the same manner as a store root file. Additionally, we may require some kind of file-global locking to go on here (not sure about this).

Read-ahead - This is probably going to be the most complicated part of the query process. Read-ahead, when done correctly, will allow the query system to anticipate the next few segments needed and prefetch ahead of time. There are two parts to this: detection of read-ahead and performing of the actual read-ahead operations. The performance part is fairly straight-forward. When read-ahead is required, the query process will just allocate and start reads going on a few more slots at a time (how far ahead to go depends on the number of queries running on the node and the number of other read-ahead streams in progress). The detection part is deceptively non-trivial, and is a problem not completely solved. There is a detailed discussion of this important topic in section 4.3.

It is important to point out here that ALL of the persistent object-specific parts of the query system are contained only within this query process. All of the other parts only contend with fixed blocks transferred to/from arbitrary files. If in the future we find a better concept for managing data that can work with a memory-mapped model of I/O, then only the query process will require changes.

#### \*\*\* 4.1.4 The shared memory manager

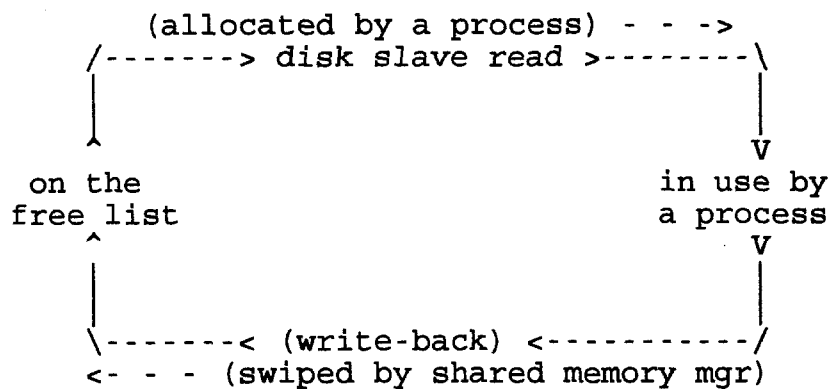
This process does not participate directly in the data path, but is instead responsible certain aspects of the data structures in the shared memory:

- 1) When any process mapping the shared slot cache on the node dies, the shared memory manager will clean up any wreckage left behind (including the freeing of any locked resources) and deallocate the dead process's PCB. It will also periodically check that connected processes are still functioning.
- 2) There is a free list in the shared slot cache from which query processes constantly allocate slots for use. The shared memory manager is responsible for refilling that free list, by stealing unlocked "old" active slots from other query processes. For optimal performance, this list should never become empty, and in fact the shared memory manager will begin scarfing up more slots when the free

list size passes below a set threshold (detection is accomplished via signal from a query process when the pool gets low). Also, there is a list of PCBs which may be waiting on the free list (when it becomes empty). The shared memory manager will wake up those processes as appropriate.

The first function is a requirement if the system is going to remain robust. Without it, then the first query process that crashes will likely leave the entire system in an unstable state. The PCB will contain enough state such that the shared memory manager will be able to reconstruct what was going on and to undo it, finish it, or just deallocate it.

The second function is part of the "circulation pattern" for slots. The pattern goes something like this:



(Fig 3 - slot circulation pattern)

When there are not enough slots on the "left" side of the picture above, the shared memory manager will start swiping slots away from various query processes until enough slots exist on the free list. Dirty slots will be written back appropriately as the operation proceeds. In a well-balanced system, the free list size should never reach zero, which means that query processes should never block waiting for a free slot.

## \*\* 4.2 Extending the big picture to multiple nodes

To make a distributed system out of this diagram, two things happen:

- 1) The single node system is replicated over all nodes, except that query processes only run on "compute" nodes, and disk slave processes only run on "I/O" nodes.
- 2) A new process type is introduced into the system, called the I/O server:

### \*\*\* 4.2.1 I/O servers

I/O server - process which forwards slot requests & data between nodes of the query system. It effectively acts as either a surrogate disk slave or a query process, depending on which side is driving the request. The I/O server process employs the system's native high speed message transport for communication among instances of itself, which in the case of the IBM SP-2, is the MPL library.

There is only one I/O server process per node, and each one is designed in an event-driven stateless style such that there is no "blocking" that

takes place when a given slot request has been forwarded over to another node (i.e. I/O server can handle an arbitrary number of outstanding requests). This is an important feature, in that it enables full overlapped I/O and potentially full utilization of the underlying switch bandwidth.

The I/O server process is going to require enough intelligence to know where to route slot requests, meaning that some kind of file name <-> node name mapping is going to be needed. In addition, when files are created, there needs to be some way to communicate that file's desired location to the local I/O server so that it may properly forward the creation operation to the correct remote end.

### \*\*\* 4.2.2 Message passing layer & C++ wrapper

The implementation of [POPM] is currently targetted for the IBM SP-2 platform, using the user-space mode of the TB-2 high speed switch system. IBM has an API, called MPL, for programming switch transfers. It is intended that an efficient class wrapper be put around this for use by the I/O server process.

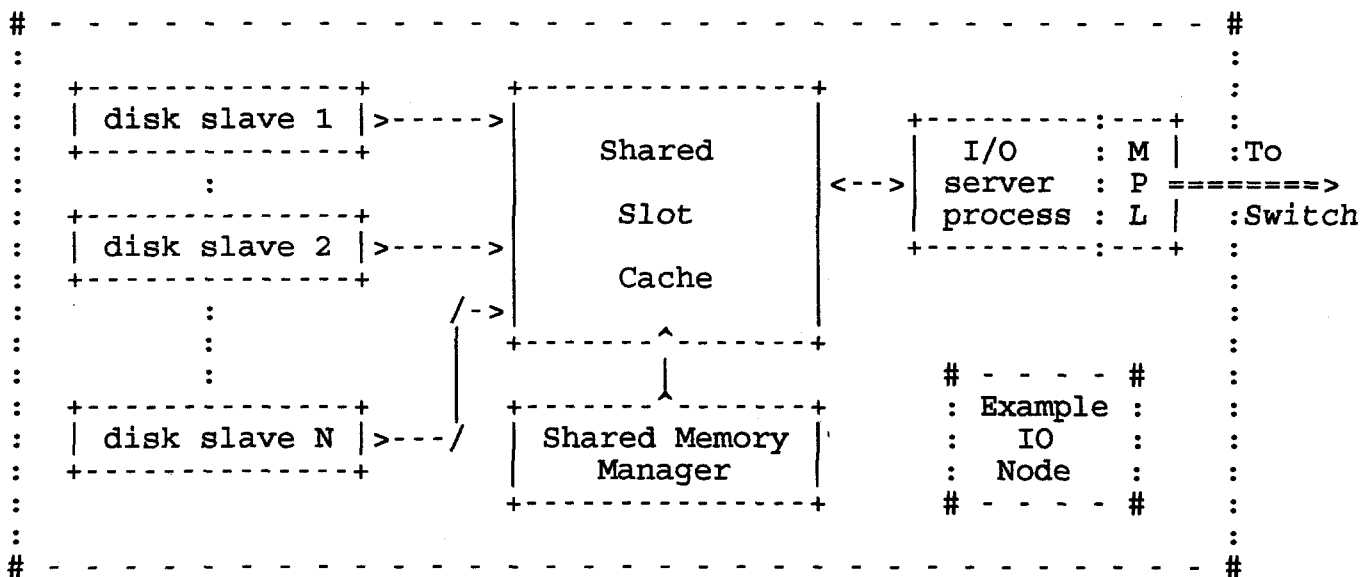
MPL has some serious performance issues that must be correctly addressed in order to achieve the maximum performance. The class wrapper should hopefully allow us to compartmentalize all of the MPL issues into one place and therefore keep the I/O server code "clean". Theoretically, then when / if we port to another system architecture (i.e. perhaps arbitrary workstations connection via ethernet!), the only changes needed should be contained within this class.

The maximum speed of a user-space programmed TB-2 switch adapter, assuming 100% CPU utilization and nothing else going on in the processor, is 30MB/sec. It appears that with the class wrapper in place we should be able to sustain effectively at least 20MB/sec between shared slot caches on different nodes.

### \*\*\* 4.2.3 Putting it all together

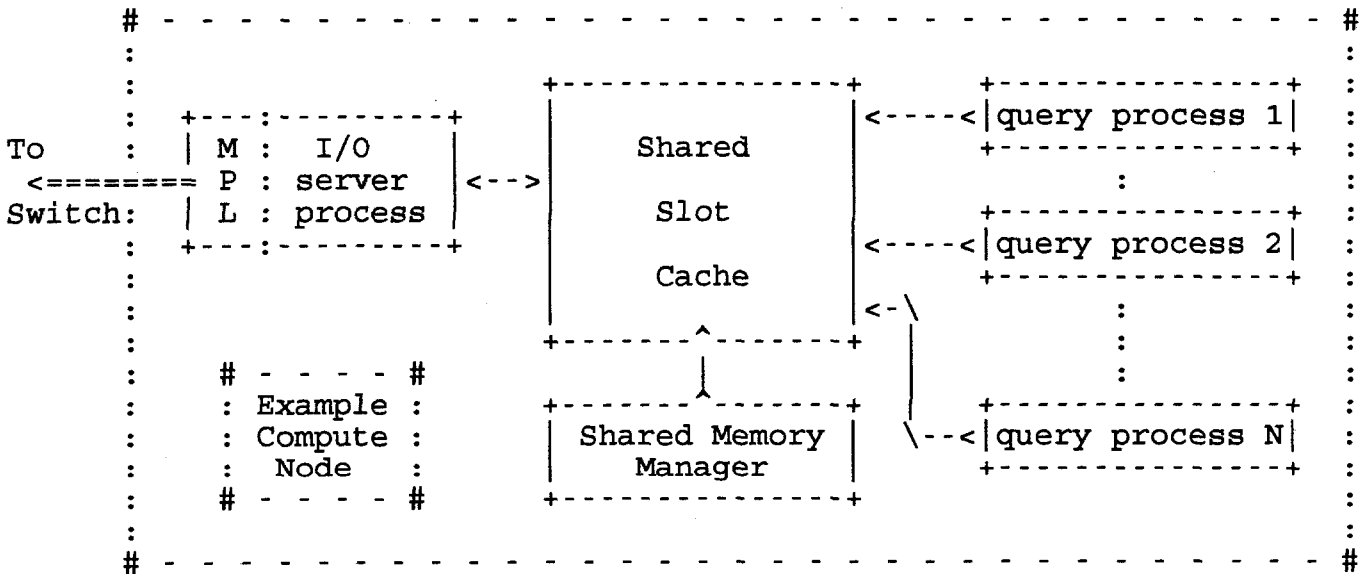
Given everything discussed so far, then it should be fairly easy now to see how the pieces fit.

Here is a diagram of the [POPM] structure on an I/O node:



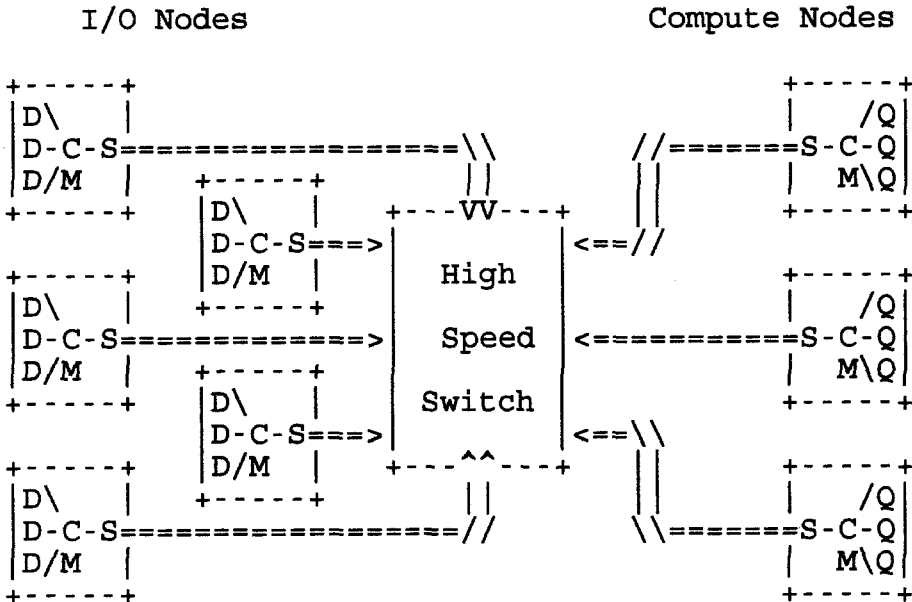
(Fig 4 - I/O node, multinode configuration)

This is the corresponding diagram for a compute node:



(Fig 5 - compute node, multinode configuration)

Here is an example 8 node SP-2 system (5 I/O, 3 Compute), with 3 disk slaves per I/O node and 3 query processes per compute node (each box is an instance of either figure 4 or figure 5 above):



```
Key:  D -- Disk slave process      Q -- Query process
      C -- Shared slot cache       M -- Shared memory manager
      S -- I/O server              = -- Switch data path
```

(Fig 6 - distributed query system overview)

That's basically it. A [POPM] query system consists of I/O and compute nodes. On each node is an I/O server interfacing the high speed switch port to the locally shared slot cache. Attached to the shared slot caches are disk slaves and query processes. Finally, each shared slot cache has a dedicated shared memory manager process for maintaining data integrity and slot distribution.

## **\*\* 4.3 Read-ahead**

In a steady-state ongoing query, the ideal situation would be a "pipeline" arrangement: as the current segment or set of segments are being scanned, the successive segments should already be on their way from disk storage. For example, while segment 0 is being scanned, segment 1 should be arriving at the compute node, segment 2 should be getting read from disk on the I/O node, and a request for segment 3 should be getting forwarded to the I/O node. Of course in reality, different stages of this "pipeline" will be running at different rates, multiple streams will be running, and multiple I/O and compute nodes will all be involved. But the idea at least should be clear.

A conventional [POPM] query process, without read-ahead, would not result in this desirable pipeline behavior. The reason is because the persistent object API models a simple synchronous behavior for segment fetching - namely that the next segment needed isn't known until the time that the process needs it. There needs to be a way to know what future segments are required before the process actually blocks on one.

The solution to this problem is to implement an algorithm in the query process which can predict what the next few segments will be, and a method in the [POPM] implementation for requesting these segments ahead of time. Also required is implementation of a concurrent segment fetching mechanism, better known as overlapped I/O. With overlapped I/O, the desired pipeline of I/O requests becomes possible. This is addressed in other sections of these notes. The real question is how does one detect and initiate the read-ahead in the first place.

Fortunately, there is one property about the nature of segment access in the CAP system that can be taken advantage of, that object stores are typically scanned in a monotonically increasing segment order. In other words, if segment N in a store is currently being examined, it is likely that segment N+K of that store, where K is a small positive integer, will be needed next. Our goal at the moment is to consider cases where  $(1/\text{avg}(K)) \geq \text{threshold}$  as read-ahead candidates, where the threshold is probably around .5 or so. A threshold of 1 means no skips, a threshold of 0 permits read-ahead regardless of how wide the average skip is. This provides a good switch for detecting when read-ahead is needed.

So, how does one define a read-ahead "stream"? Assuming that a single store will contain all the objects of a given "type" that are being sequentially scanned, then the read-ahead parameters can be maintained as extra state when the store is opened. Whenever a segment for a given store is asked for, a check for K=1 is performed. If the K value is generally not "1", then read-ahead is disabled. If K=1, then read-ahead is employed, and extra requests are generated for the next few segments in sequence. That's the easy part.

Note that if several logical "streams" are being scanned within a single store, then the above technique for read-ahead detection will fail. In that case, some help will be required from the API to enable read-ahead.

Once it is decided that a given stream needs read-ahead, how far should the read-ahead go? That is a much more difficult question to answer. The ideal answer is to read-ahead just enough to keep the query process from ever having to block waiting for a not-yet-present segment. Reading segments ahead any further than that amount (well maybe a few extra segments just to be safe) should not have any appreciable additional

performance impact.

Implementing this ideal solution is not very easy. In theory, the query process will have to measure its own average scan time per segment and compare that to the average apparent latency for fetching a segment. The read ahead depth should then be 1 plus enough extra segments to bury enough latency such that the remaining latency is less than the scan time. If the scan latency is already higher than the fetch latency, then the computed read-ahead depth of 1 should be sufficient.

In practice, the above idea may be too expensive to implement, in which case something more primitive, but less accurate could be employed. For example, a simple feedback mechanism could be tried: increase the depth whenever segment N+1 is not present when finally needed, and decrease the depth whenever segment N+2 is already present when segment N+1 has been read in time.

In addition to measuring and reacting to segment latencies, the read-ahead depth is also limited by resources available on the compute node. Whenever a read-ahead stream has been enabled on a compute node, a stream count is incremented in the process's PCB. The total of all read-ahead stream counts in the compute node is used to ration the overall use of slots. In other words, if only a single read-ahead stream were in progress, its depth is only limited by the number of slots on the node. With more streams active however, the slots will be divided among the streams, thereby limiting the maximum depth for any particular stream.

Very interesting note: The definition of an ideal read-ahead depth above hinges only upon whether or not the query process might block waiting for I/O. One might notice that this has nothing to do with whether or not all of the I/O nodes have to be kept busy. It is entirely possible that only a fraction of the overall I/O bandwidth would actually be needed to keep a query occupied. The fraction required is therefore driven purely by the speed of the query process. This is actually what we want! Remaining bandwidth will therefore be available to other queries in the system. If not enough total bandwidth is available, the result should be a fairly even division among all of the competitors. How about that: automatic bandwidth allocation!

Note also that even if full read-ahead can't be achieved in a given instant, partial read-ahead is still very beneficial. Every concurrent segment fetch in progress is that much less latency for which a query process must block.

Much more thought is still needed for an efficient read-ahead implementation in [POPM].

## **\*\* 4.4 Other important details not yet covered**

The next set of topics deal with the remaining issues about the query system that have either not been covered, or require more detail for proper understanding.

### **\*\*\* 4.4.1 Efficient slot cache searching**

The UIC ptool implementation performs a linear search when locating a slot cache entry. Though the search starts at the point where the last "hit" was made, resulting in minimal search time for consecutive dereferences of the same segment, this degrades to worst case performance

if two different segments are being alternately dereferenced.

If [POPM] were to employ this algorithm, significant problems would result. [POPM]'s slot cache size is not limited to a small fixed number, and if it gets large (which is entirely likely if only a single query is happening on a node) the search time overhead will become unbearable. The solution involves the use of multiple chains of slots with a hashing function:

- 1) Slots in [POPM] may have arbitrary 64 bit tags assigned.
- 2) When a tag is assigned, the tag is hashed to a 5 bit number, which selects one of 32 possible PCB-specific linked slot lists. The slot is placed in the linked list to which it is hashed.
- 3) When a search is conducted, the search tag is hashed using the same function to produce another 5 bit index into the array of linked slot lists. The correct list is selected and a linear search continues from that point.

With a reasonable hashing function (i.e. one that results in an even distribution), the number of slots to check for a given search will go down by a factor of 32. If that isn't enough we can try a larger array, like 64 or 256 linked lists. The goal is to reduce the length of the linear search to only 1 or 2 actual checks.

#### \*\*\* 4.4.2 Locking issues during dereference

Since it is possible for a slot to be stolen asynchronously from a [POPM] process (by the shared memory manager), then it is possible for the hashed array of linked slot lists to be changed asynchronously. This causes a major problem:

- 1) The hashed array must remain stable while it is being searched, which implies the use of a locking mechanism. But...
- 2) The searching algorithm which must acquire this proposed lock is going to be in the critical path of the persistent pointer dereferencing algorithm. Therefore the lock must be extremely fast, preferably inline code!

Normal SYSV-style semaphores therefore won't work. They're just too slow; every call results in a trap to the system kernel. The AIX operating system has a faster semaphore primitive which avoids a system trap if the lock is free, but it can't be inlined.

The solution is the application of the well-known Dekker semaphore algorithm. The implementation requires no special hardware primitives, can be inlined easily, and is very fast if only two processes are contending for the resource (the algorithm gets more lengthy if more contenders are possible).

This issue is the reason why slots are allocated to a PCB instead of always being shared among all PCBs. If global sharing were allowed, then the hashed array of linked lists would also have to be global. This means that the number of contenders for it could be as many as there are PCBs in the system, which makes the Dekker algorithm a practical impossibility. In addition, with more contenders, then this resource is likely to become a global bottleneck.

Since the hashed array of linked lists is private to a PCB, then there are only two contenders - the process owning the PCB, and the shared memory manager. This makes the Dekker algorithm an ideal solution.

An example implementation of the Dekker semaphore algorithm is shown in figure 4.7, page 86, of the book "An Introduction to Operating System", by Harvey M. Deitel (Addison-Wesley, 1984).

### \*\*\* 4.4.3 Implementation for physical pointers

A physical pointer is a pre-dereferenced persistent pointer. An instance of a physical pointer has a pointer into the actual slot where the data currently resides. Since the shared memory manager is allowed to steal slots from a process, then how is it possible to guarantee physical pointer integrity?

There are two possible ways to achieve this; both have advantages and disadvantages. One method involves back pointers, the other uses a reference count.

The back pointer method:

In this case, any physical pointer instance which points to a given slot becomes a member of a back pointer list (array, container, whatever) associated with that slot. Logically, now it is possible to find out which physical pointers are currently referencing a slot: If the slot is stolen, the list is traversed and all referencing physical pointers are found and invalidated. To invalidate a physical pointer, its internal "real" pointer is set to zero. The next time the physical pointer is used, this null value is noticed by the internal physical pointer class implementation, and the original persistent pointer value is re-dereferenced to find a new slot.

This has one major advantage in that it is impossible to cause a resource deadlock. Having a physical pointer into a slot does not imply any kind of lock; at any time the slot may still be successfully swiped. If there are too many physical pointers in use, the query performance will just degrade, down to a level equalling that of not using the physical pointers in the first place. However, there is no overall limit on the number of physical pointers.

There are two disadvantages. First of all, since the slot is never really locked down, it is possible for the slot to get stolen in the window of time between when the "real" pointer is checked for a null value and when it is used. This is a serious error, which will become more clear in the next section about pointer lifetimes. The other problem is worse; the physical pointers are stored in the private data of the query process, making it impossible for the shared memory manager to access them for purpose of invalidation. Now, it is possible to store the physical pointer structures in the shared memory with the other data structures, but then that poses a hard upper limit on the number of physical pointers that a process can use - which effectively eliminates the only advantage.

The reference count method:

Each slot has a reference count integer in the control information, which is initialized to zero. When a physical pointer is assigned, the pointed-to slot's reference count is incremented. When the physical pointer is cleared, destroyed, or pointed elsewhere, the reference

count is decremented. So long as the reference count is non-zero, the slot is considered "locked" and immune from being stolen by the shared memory manager (or otherwise reused). Slots owned by a process with a reference count of zero are open to reclamation by the shared memory manager.

Guaranteeing integrity of the reference count is not nearly as difficult as it may seem. First of all, the physical pointer type is implemented as a class, with constructors and destructors and full syntactical protection - which means that there is no burden on the user for keeping the reference count correct. Instead the physical pointer implementation just adjusts the reference count as appropriate when things change. Second, physical pointers may only point at slots owned by the process. A physical pointer may not be "aimed" at one of another process's slots, so it is impossible for one process to interfere with another's slot reference counts. This also means that: a) When a slot is allocated to a process, that slot's reference count automatically starts at zero, and b) if the process crashes and wreckage has to be cleaned up, the reference counts for slots owned by that process become irrelevant.

The big advantage of this scheme is that it clearly defines the lifetime of a physical pointer's "real" pointer. So long as the physical pointer is pointing at a given slot, that pre-dereferenced "real" pointer is guaranteed to remain valid - no races.

The disadvantage however is critical: use of too many physical pointers can cause deadlock on a node. When a physical pointer is referencing a slot, there is simply no way to involuntarily break that relationship (well we could kill the process and do a wreckage clean-up...). A poorly written user query could employ too many physical pointers, potentially causing a slot resource deadlock.

We can actually minimize any impact from this disadvantage, since the shared memory manager will be able to detect how many slots a process has locked. An artificial threshold could be implemented - processes which exceed the limit could just be killed. Another less drastic approach is the for the query process itself to keep a count of active physical pointers and simply cause an error if too many are created.

The [POPM] system will use the reference count method in the physical pointer implementation. In reality, there will be two classes:

- An underlying class for creating and maintaining a reference lock on a given segment. It is at this layer where the reference count integrity will be maintained.
- A physical pointer class, which will contain an instance of the above class either as a base class or a member.

This separation of classes and use of the reference count method enables a solution to a subtle problem having to do with persistent pointer lifetimes, discussed in the next section.

#### \*\*\* 4.4.4 Managing persistent pointer lifetimes

There is an insidious subtle problem with the present persistent object API having to do with the lifetime of a dereferenced persistent pointer. This flaw in the API definition is currently not causing problems in the use of the UIC ptool product only because of a fortuitous feature of the

implementation, and because current applications tend not to stress cases where many persistent pointers need to be active at one time.

Here's the scenario: A line of code assigns an int field from one persistent structure to the value from another persistent structure. This line of code is going to generate two dereference operations, one for the l-value, and one for the r-value. Assume that the two persistent structures point to different structures and that the slot cache is only one element in size. Now, if the compiler generated code like this:

```
int *tmp, v;
tmp = dereference(rp_ptr); /*1. Dereference r-value*/
v = *tmp;                  /*2. Get r-value*/
tmp = dereference(lp_ptr); /*3. Dereference l-value*/
*tmp = v;                  /*4. Assign l-value from previous r-value*/
```

Then things would be fine. But suppose instead the compiler generated this sequence:

```
int *rtmp, *ltmp;
rtmp = dereference(rp_ptr); /*1. Dereference r-value*/
ltmp = dereference(lp_ptr); /*2. Dereference l-value*/
*ltmp = *rtmp;              /*3. Assign l-value from r-value*/
```

Now there's a fatal problem, because when step (2) took place, the slot previously assigned for the r-value pointer in step (1) will have been remapped to the segment for the l-value pointer required in step (2)! The assignment will read random data. This sequence is a perfectly legal interpretation of a C expression, because except for the case of the boolean logical operators, neither C nor C++ stipulate in what order or at what points functions within an expression must be called!!

This case was contrived, but only to illustrate a point. In fact, the UIC ptool implementation survives this problem because the slot cache is larger than one element, the replacement algorithm is LRU, and few applications ever need many simultaneous persistent pointers to be valid at a single instant. With LRU replacement, the number of persistent dereferences in an expression would have to exceed the maximum size of the slot cache before trouble could happen.

The whole problem comes down to defining the required lifetime of a dereferenced persistent pointer. (This issue actually has broader implications in the ongoing C++ standardization effort because the same problems erupt when defining temporary class instance lifetimes). Basically, the lifetime of an "element" depending on a temporary variable generated by the compiler must persist at least until the next "sequence point" in the source code (at least a line of code).

In the case of [POPM], two further complications arise. First, the slot cache size can dynamically change, so there is absolutely no way to assume a safe upper limit on the number of dereferences between sequence points. But the really bad problem is that an asynchronous process (the shared memory manager) is able to steal slots - it could just as easily steal the slot just used as part of a dereference but before the resulting "real" pointer is used (but not likely since we are also intending on using LRU slot replacement). This a dangerous problem that must be properly dealt with.

It turns out that the solution to these issues in [POPM] are simple, when the techniques used for physical pointers are borrowed. Notice that

physical pointers (as implemented with a reference count) don't have any pointer lifetime problems because in that case, the slot is explicitly locked in place when the physical pointer is aimed at it. Obviously, one could disallow the dereferencing of persistent pointers altogether and only allow access through physical pointers. But that would make the persistent object API no longer backwards compatible with the UIC ptool implementation. There is still a middleground: Use a global instance of the physical pointer's base class. This class is able to lock a slot. So now whenever a new persistent pointer is dereferenced, instead use the global slot locking instance to grab the slot and then get the dereferenced value from there. The slot is therefore guaranteed to stay valid (and therefore the dereference result stays valid) until the next persistent pointer dereference operation (at which point the global slot locking instance is pointed elsewhere).

This solution guarantees at most one valid regular persistent pointer dereference between sequence points, which is still not good enough. But the solution can be extended. Instead of a single global instance of the physical pointer's base slot locking class, use a circular list (implemented as an array with a rotating index). Each successive dereference operation uses the next slot locking instance in line. For a circular list of size N, then the last N dereference operations are guaranteed to remain valid between sequence points.

NOTE! The size of this value N (i.e. the maximum legal number of dereferences between sequence points) should be defined as part of the persistent object API. Currently it is not. This is a flaw in the current API and needs to be discussed. Making it larger is generally good, but it can waste resources if it is too large, since the last N dereferences are going to cause (up to) N slots to always remain locked. The fact that current implementations are not having these problems in a serious way indicates that N=8 (size of the slot cache in the UIC ptool implementation) is a good starting point for evaluation.

#### \*\*\* 4.4.5 Allocating PCBs

When any process in the [POPM] system starts and maps the local area of shared slot cache, it must immediately allocate a PCB (Process Control Block). A fixed size array of these structures exist in the shared memory. Allocation of a PCB is important in that it provides for:

- 1) Tracking by the shared memory manager. Enough information in the PCB exists to find the associated process. In addition, enough state is contained in the PCB such that if the corresponding process crashes, the shared memory manager will be able to undo any half-completed operations, finish irreversible operations, and free any resources / locks held by that process (including the PCB itself).
- 2) Unique identification in the [POPM] system. The logical process id of that process is the element number of the PCB that gets allocated.
- 3) Resource allocation. Any resources in the shared memory allocated to the process (i.e. slots) are represented in the PCB, and various overheads to maintain that allocation are instanced by that PCB.
- 4) Communication. Any explicit communication among processes may be carried out via structures in the PCBs.
- 5) Locking. Things which get locked to a process are represented in the PCB of the locking process.

- 6) Synchronization. The state of the owning process (i.e. running, blocked on free slot wait, performing look-up, idle, etc) is always represented in the PCB. When the process has to block for some [POPM] related reason, the PCB itself will be put on a list representing that blockage - which allows other processes to find the blocked process.

So it should be a pretty safe conclusion that being able to unambiguously allocate PCBs is a good thing. But how does one do this in a bullet-proof way? All other forms of allocation in [POPM] are without holes because the PCB allows for tracking of the allocation. But when the PCB is being allocated, who tracks that? And how does one allocate a PCB without first having to acquire a lock to guarantee atomicity during the allocation? Remember that any number of processes may be trying to allocate a PCB at one time - there needs to be a way to serialize this, and just acquiring a global exclusive lock opens up a huge hole if the holder of the lock should crash before releasing the lock.

The answer is incredibly simple: Use the Unix pid as the allocation flag in combination with the AIX "compare and swap" (cs()) atomic operation. A field in the PCB exists (call it the upid) to contain the actual Unix pid of the owning process. When the upid field is zero, the PCB is considered to be free. When it is non-zero, it is owned by the corresponding Unix process. This is ok because it is impossible in Unix to have a pid of zero.

Now, to allocate a PCB, first perform a linear search and find a PCB with a upid value of zero (i.e. free). If nothing could be found, then all PCBs are in use. If a free PCB is found, then perform the compare and swap primitive operation on the location with a compare value of zero, and the process Unix pid as the swap value. If the compare fails, then another process got the PCB, and the algorithm should restart the PCB search. If the compare succeeds, the new Unix pid will be written into the upid field and the PCB will be officially allocated!

The beauty of this scheme is that the instant the upid field has been written, it is possible for the shared memory manager to locate the owning process. That process could fail on the very next clock cycle without causing any problems in the system - the shared memory manager would eventually discover the dead process and just free the PCB again.

This solution provides for a bullet-proof, no-window, holes-free means for allocating PCBs. Once a PCB has been allocated, then it is possible for that process to allocate other shared resources & locks in the shared slot cache without risk of causing resource "leaks" or orphans should it crash later on.

Note: If this system is implemented under Irix, then the ucas() function may be employed to take the place of the AIX specific cs() function for compare and swap. This is available in (at least) Irix 5.2 and up.

#### \*\*\* 4.4.6 Cleaning up wreckage

This concept was mentioned in the previous section. The [POPM] implementation has a shared region of memory, which leaves open the possibility that malfunctioning processes could corrupt the data structures contained therein and cripple the overall system. This is an important point because the [POPM] servers are designed to stay up effectively forever, while [POPM] query processes, which have linked-in user code, will come and go over time. One user bug which causes a

[POPM] query process to crash should not be able to leave the shared data structures in an indeterminate state and cripple the rest of the system. How is this addressed?

There are three types of problems that are possible within the shared slot cache:

- 1) Crashing while not doing something in the shared slot cache. When this happens, there may be resources allocated to that process that need to be freed.
- 2) Crashing in the middle of an operation in the shared slot cache. In this case, one or more data structures relevant to the operation in question may be left in an inconsistent state.
- 3) Random memory scribbling / wild pointer dereferencing (not persistent pointers, these are "regular" pointers). In this case, user code has malfunctioned such that incorrect memory locations are being overwritten. Some of those locations could fall within the mapped address space of the shared slot cache.

When a process crashes, the shared memory manager will eventually figure it out (i.e. periodic kill(pid,0) operations), and take recovery actions.

For issue (1), the action is simple since all allocated resources are tracked in the PCB - the shared memory manager need only free the resources, and then free the PCB.

Issue (2) is somewhat more complicated, but can still be dealt with. The trick is that whenever any process begins an "atomic" operation of this nature in the shared memory, it should write enough state / status information into the PCB such that the shared memory manager can discover what was taking place. Then it can either undo or complete it, depending on the nature of the operation. Following that, the shared memory manager would complete the recovery by following the steps for issue (1).

Issue (3) is problematic. There is no bullet-proof defense against it. The problem is that certain control structures simply need to be read / write accessible to every process in the system. Slot data not logically writable to a process can be protected by either unmapping or write-protecting it. The control structures can't be protected in this way. However there are three mitigating factors: 1) The aperture of "vulnerable" shared memory is small compared to the size of the process context in which a wild pointer may mess up. Assuming even distribution, then it is unlikely that a wild pointer will actually corrupt the shared memory control structures. 2) Most of the [POPM] user code is generated by another program, and such code is usually resistant to wild-pointer errors resulting from bad input. 3) Any kind of wild pointer that DOES happen to corrupt a control structure will result in the entire system crashing - something very obvious and unambiguous. Since slot data can't get accidentally changed by such an error, the failure mode of just getting a wrong answer with no indication of a problem simply can't happen.

Warning note: Any place where the shared memory manager is contending for a resource or otherwise waiting for some operation from another process should NEVER block. There must be an escape hatch if the other process crashes. The shared memory manager should still periodically check all processes. If a process upon which it is waiting on dies, it should stop waiting and take a recovery action!

#### \*\*\* 4.4.7 Managing multiple address spaces

A persistent address space is defined by the "dbmap file", which in turn defines all of the stores, which in turn defines all of the folios. To permit multiple address spaces in a single system really only requires two things:

- 1) Support for multiple dbmap files.
- 2) Support for multiple "file spaces". Stores and folios within an address space should only need to be uniquely named inside that particular address space.

Requirement (2) is trivially achieved in [POPM] by using directories. In fact, the address space's name can simply be the directory path required for locating it. In a distributed environment, each I/O node should conceivably implement the same path to its share of that particular address space. Presumably all address spaces (i.e. directory paths) would start from a common point, which itself can be defined local to the particular I/O node.

With requirement (2) addressed, requirement (1) becomes equally trivial. The dbmap file name is a constant, and simply lives in the directory of the address space.

Note that since address spaces are defined by directory paths, the ownership and permissions of that directory can be applied when validating user access into the particular address space.

When a [POPM] application process starts up, no stores can be opened until the address space has been declared. Given the above, then that operation only requires setting the directory path. Once set, the path is simply prepended to all file names generated by the [POPM] application process.

#### \*\*\* 4.4.8 Memory mapped I/O

One important aspect of this system that bears emphasis is the nature of the I/O access model. Even though explicit I/O is being performed, the model is that of memory mapping.

Notice also that everything having to do with the persistent object API sits above the I/O access model. The servers and disk slaves are totally generic. This an important distinction: All the concepts of segment, folio, store, address space, etc are only in the query process. Below that point is simply an API for mapping 64KB chunks of arbitrary files from arbitrary nodes.

In fact, one can view the entire I/O model as being sort like another kind of network file system. It only serves to implement a demand-mapped distributed file-I/O system. Except in this case, the granularity of allocation is 64KB (the segment), and the access method more resembles mmap() to a fixed area of memory, instead of read()/write().

#### \*\*\* 4.4.9 Store root file metadata access

This question may have occurred while reading the previous section: If the entire method for accessing files is through the slots, then how is the store metadata stored? Answer: The store metadata is contained in a

root file, which coexists with the folio files. Access is accomplished by mapping the root file as an ordinary segment, using the same mechanisms available for accessing segments within a folio. The root file should be small enough to fit in an entire segment slot.

There are a few "details" to pay attention to here:

- 1) If multiple queries are operating on the same store(s), then root files may be mapped multiple times. This may be a problem if the store metadata is going to be changing. However once the store is created, the metadata should be treatable as read-only data. If this is not true, then some kind of file locking mechanism will have to be added to the [POPM] I/O model.
- 2) Slots are expensive. If store metadata needs to be kept around for a while, the slot used to map the root file should not be kept indefinitely (i.e. by keeping a physical pointer aimed at it). Instead, the metadata should be immediately copied out to process-private heap storage and the slot released for reuse. Else if many stores are kept open, each one is going to permanently occupy a slot and the possibility exists that all the slots will be used up just to retain store metadata.

#### \*\*\* 4.4.10 dbmap access

The "dbmap" file is the entity that is used to coordinate allocation of store numbers among the stores of a given address space. It contains a mapping from store number to store name. It may also be used to contain address space-global metadata. For a given address space, there is only one dbmap file.

A query process needs access to the dbmap file. This can be accomplished by using the same solution as that applied for root file access: map it into a slot. But there are two important differences:

- 1) The dbmap file size can exceed the size of a single slot, since it can grow as more stores are created (and more store numbers have to be allocated / tracked).
- 2) The potential exists for concurrent writes to the dbmap file from different query processes. This (in theory) shouldn't happen with root files, since they are written once when the store is created. But the dbmap file gets a new entry for every store that is created. If multiple processes are creating stores, then concurrent writes to the same "segment" of the dbmap file is possible (and likely).

The first difference is not really a problem; it is solved by just paying attention to the possibility when designing the file format and the code that needs to access it.

The second difference however is a much bigger problem. We must either (a) implement some kind of record locking mechanism, (b) disallow the creation of more than one store at a time (i.e. only one process in the system may create stores), or (c) use an entirely different method for dbmap file access. The first choice is the "cleanest", but also the most difficult. The second choice can be done without too much pain, given our intended use of the system. The third choice is repugnant.

#### \*\*\* 4.4.11 Searching for files in a distributed address space

This issue is directly relevant to the issue mentioned in section 4.2.1 of mapping file names to nodes. When a given compute node has a request to map block X of file "abc", how does the I/O server know which I/O node the request should be directed?

There are two answers. First the easy one. If the file in question is being created, the query process must communicate the node name. This is because the I/O server has no foreknowledge about where new files should be created. The query process however knows the folio striping algorithm from the store metadata, so it can compute what the correct node should be and can communicate that to the I/O server as part of the request.

Now the tough answer. If the file in question already exists, then the I/O server must discover the correct I/O node. Logically, this can be done via a search of all the I/O nodes, but this is horribly inefficient. Each I/O server should probably maintain a file directory cache from each I/O node. When a file is asked for which is not in the cache, the I/O should then perform a search to update the cache. Coherency can be maintained among the I/O server processes. Here are the details:

- 1) Each I/O server should maintain an in-core cache / hashed table of file name to I/O node mappings.
- 2) The cache starts out empty; when files are asked for on a compute node, the local I/O server first checks its cache. If there's a "hit", the request is just forwarded to the I/O server on the indicated I/O node. If there's a "miss", all I/O servers on all defined I/O nodes are asked about the file. The response is used to create a new cache entry, and the request is then directed to the responding node.
- 3) I/O servers can exchange unsolicited information about cache entries. This is used both for maintaining coherency among all the caches and for efficiency reasons. For example, if a file is deleted, the I/O server on the I/O node where the file got deleted should propagate that information to all compute node I/O servers where a cache entry needs to be invalidated. Also, when an compute node I/O server requests information on a particular file, I/O node I/O servers may respond with multiple entries (i.e. an entire directory's worth).
- 4) The cache size should be fixed in size (hundred's of entries), N-way set associative, and should use a hashing algorithm for look-up. Older entries are expunged when room is needed for newer entries.
- 5) I/O node I/O servers should remember to which compute nodes they have sent information. In that way, these servers will know where to send updates if the information changes.
- 6) There needs to be a way for a local I/O server process to get a directory listing. This is not as obvious as it sounds, since the associated disk slave might NOT be using physical disk storage - they could be talking to an HSM for example. Therefore the directory listing operation should be part of the disk slave implementation, which suggests that a way is needed for an I/O server to communicate this information to a disk slave....

This scheme is probably more complicated than one would like, however it does provide for a smooth illusion of a homogeneous file space distributed across all of the I/O nodes.

### \*\*\* 4.4.12 Store deletion

Up to this point, there has been no thought given toward distributed I/O operations that fall outside the primitives of merely mapping / unmapping segments of files. For example, what about file deletion & renaming?

The renaming issue is simple: It isn't an issue. If a file need to be renamed, this can be accomplished through manual intervention, as the persistent object API has no requirement for this ability.

The deletion issue is more involved. In a running production system, it is intended that over time stores are to be created / migrated into the system, and then expunged at some later time, to be replaced by other stores. Though store deletion isn't part of the persistent object API, it is something that will happen often and should be easy to perform. Remember that store deletion requires removal of the root file, all folio files on all I/O nodes, and removal of the entry in the dbmap file. This should therefore be considered. So here's some thoughts:

- 1) A store delete operation can be created in the persistent object API. This would be translated into a series of file delete requests to the distributed I/O system, which can be communicated to the I/O nodes via a communications interface in the PCB (there is no natural way to do this in a slot).
- 2) Store deletion could be accomplished through some mechanism external to [POPM]. The only caveat here is that the delete operation is going to write to the dbmap file, which will require some locking protocol between [POPM] and the external deletion mechanism.
- 3) Store deletion is disallowed. The only way to erase a store is to erase the entire address space. Cleaning out an entire address space is easy: just do a global delete of all files in the address space's directory.

It is not clear right now what is the right thing to do. This needs further discussion.

### \*\*\* 4.4.13 Storage classes

The query system pictured so far has only disk slaves on the I/O nodes and query processes on the compute nodes. The I/O server processes would act as go-betweens: on the I/O nodes they act as surrogate query processes, while on the compute nodes they act as surrogate disk slaves.

Simple enough, but what would happen in this system if a query process were run directly on an I/O node? This is a problem, because when that query process requests slot I/O, the request is going to go directly to the local disk slaves, bypassing the I/O servers and therefore making it impossible to do any off-node I/O.

Another related situation is possible if a single I/O node were to (attempt to) support both physical disk storage and HSM storage. This would mean effectively two different types of disk slave processes coexisting on the same node - there would be no way to properly direct the request to the right subset of disk slaves. In the current proposed implementation, the I/O request would just be picked up by whatever disk slave saw it first, which is wrong.

The solution to the first problem is to be able to tag requests as local

or global, and to have separate I/O queues in the shared slot cache for remote requests. If the request is local, it is placed in one of the local request queues (read, read-priority, write). If it is global, it is placed in a global request queue. The I/O server only watches the global queue in the shared memory. Upon receiving a request, it is forwarded just like before. Except if the destination node is really the local node, the treatment is simple: just requeue the request to a local request queue.

The solution to the second problem is to implement the concept of a "storage class". Slot I/O requests are tagged with a storage class, which in the global case is used as additional forwarding information and in the local case is used to choose which subset of disk slaves will be performing the service. Example of storage classes include physically, and HSM. Disk slaves which perform physical disk I/O would only service physical-class requests, while HSM-serving disk slaves would only service the HSM-class requests.

Efficient implementation of multiple storage classes along with local versus globally tagged requests would just require multiple sets of queues in the shared slot cache for pending I/O:

- A set of queues (read, read-priority, write) for physical disk access.
- A set of queues (read, read-priority, write) for HSM access.
- A queue for global access (in which case the I/O server would forward the request to the correct destination and drop it into the appropriate queue at that end).

When a query process needs to perform I/O, it would set up the request and then attach it to the appropriate queue (and signal any process waiting for I/O on that queue).

Note that the choice of actual storage class could also be made in the I/O server process by imposing a convention for the path of the file name being requested. For example, any path name beginning with "/hsm" would indicate HSM class access, in which case the I/O server on the I/O node would direct the request into one of the HSM-class request queues. This approach would make storage class determination more transparent to the query process (but would also complicate the multiple address space implementation).

#### \* 5 Example of a persistent pointer dereference

Here is the series of steps taken to dereference a persistent pointer. This assumes a distributed configuration (i.e. I/O servers are being used), and no read-ahead.

Key:  
+----Q = Query process  
| +---C = I/O server on compute node  
| | +--S = I/O server on I/O node  
| | | +--D = Disk slave  
| | | |  
QCS D

1. Q--- Query process dereferences the persistent pointer.
2. Q--- The persistent pointer is converted to a 64 bit tag, by zeroing

the offset portion of the pointer. This is then hashed to form an index into an array of linked lists (residing in the PCB, currently 32 elements, requiring 5 bits of index).

3. Q--- The correct linked list, which is selected according to the hash index, is linearly searched. The search is accomplished by comparing the tag computed in the previous step with the stored tag of each slot in the list. If a match is found, the offset of the persistent pointer is added to the virtual address of the slot, and the algorithm is complete. If the tag is not matched, then the remaining steps must take place:
4. Q--- The query process allocates a new slot from the globally shared free list. If the list is empty, the query process puts its PCB on the global list of processes blocked waiting for slots, sends a signal to the shared memory manager, and waits for a signal. As soon as it wakes up and finds its PCB off the blocked list, it restarts the attempt to get a free slot.
5. Q--- The query process initializes the slot's tag with the tag computed in step 1 (which also computes the hash and inserts the slot in the appropriate hash list), and fills in the file name & block number in the control information. The file name and block number are computed by using the store metadata, which is found by looking up the store number. The slot is then placed on the "read with priority" list of slots, the first idle server process is found by pulling a PCB off the list of idle servers (which in this case would be the I/O server), that server is signalled, and then the query process waits for a signal.
6. -C-- The compute node I/O server notices the pending request, and determines the correct I/O node for the request (via lookup in a file to I/O node cache maintained among the I/O servers - see section 4.4.11).
7. -C-- A request message is composed and sent through the network to the I/O server process on the target I/O node. In the case of this operation, included in the message is the file name and block number of the request. The I/O server also provides information in the message so that the incoming segment can be read directly into the slot data.
8. --S- The message is received by the I/O server on the I/O node. The server uses the message information to allocate a slot and set up a local request. The algorithm for allocation is the same as that for the query process back in step 4. The request is placed on the "read with priority" list of slots, the first idle server process is found and signalled.
9. ---D A local disk slave wakes up, sees the request, and performs the operation. Upon completion, it signals the owner of the slot (the I/O server).
10. --S- The I/O server process on the I/O node notices the completed request and forwards the slot data and result status back to the I/O server on the original compute node.
11. -C-- The I/O server on the compute node receives the data for the segment directly into the slot. The status is updated, and the owner of the slot (the query process) is signalled.

12. Q--- When the query process wakes, it checks the slot for I/O completion. When the I/O is complete, the process checks for success - if there is a failure, then the query must exit with an error. On success, the query process adds the persistent pointer offset to the virtual address of the slot, and the algorithm is complete.

The above steps do not include anything about resource locking, slot reference count updates, or updating fields required for implementation of slot aging.

Notice that the I/O servers are able to handle multiple concurrent requests. The internal implementation of those servers is a "work list" which is constantly being processed and added onto. As new requests show up and messages are received, work gets added. The server constantly pulls items off that list and processes the required work. Each of the steps above involving an I/O server is independant of other requests. Once such a step is completed, therefore the I/O server is free to find other requests needing work at other phases of the transfer.

#### \* 6 Expected performance improvements

Listed here are the features / functions of [POPM] that should result in performance improvement over the currently implemented Fermi-ized ptool.

The overall performance improvement that should be observed is that resulting from the ability of [POPM] to execute overlapped I/O - from the physical storage all the way out to the query process. With full overlapped I/O comes far better utilization of all of the resources along the I/O path, specifically including the communications network (i.e. 30MB/sec at 80% duty cycle instead of 10% duty cycle).

The next few sections outline various parts of [POPM] that make overlapped I/O possible.

#### \*\* 6.1 Shared slot cache

As said before, the slot cache is now a structure physically shared among multiple query processes. This has the following benefits:

- +Copy operations are minimized. Reads & writes from server processes go directly to the slot. In a single node query, zero memory to memory copies are needed, while in a multi-node configuration, the number of copies is theoretically the minimum possible.
- +Memory contention among multiple query processes is handled better, since now they can dynamically share one pool.
- +Performing read-ahead is now just a matter of setting up a few more slot cache entries for pending segments.
- +Pending I/O operation state is completely containable within a slot (and its companion control information). This means that on a given node, there can be as many concurrent I/O operations as there are slots available.

#### \*\* 6.2 Read-ahead

A good read-ahead algorithm makes it possible for [POPM] to create more

usable I/O operations for a given query process. Without read-ahead, a single query could at most cause only 1 pending I/O operation at a time. With read-ahead the next few segments can be scheduled concurrently, thus overlapping the I/O reads and burying the latency.

### \*\* 6.3 Segment & folio striping

The ability of striping segments (or folios) over multiple folios (or files) does not in of itself result in overlapped I/O and better performance. What it does do is permit better read-ahead. Without this striping, the read-ahead distance would have to be much larger before any parallelization of I/O can occur. With the striping, the read-ahead distance need only be a few segment's worth.

### \*\* 6.4 Distributed I/O effects

This point is obvious, and was one the reasons for the previous creation of the Fermi-derived ptool in the first place. The I/O disks are spread over multiple nodes, thus spread the I/O load across CPUs and increasing the number of parallel I/O operations. In [POPM], this organization should have a "synergistic" effect with the other enhancements in place.

### \*\* 6.5 Stateless requests

The only state describing a request is contained in the control data for the slot in question. This results in far better resource usage along the I/O path. For example, the I/O servers don't really have to "remember" anything about a pending request to an I/O node and can more easily go about finding and forwarding other requests while that first request is proceeding. Result: overlapped I/O.

### \*\* 6.6 Multiple disk slaves

There is no specific limit on the number of disk slave processes that may be running on a single node. They're sort of like nfsd daemons - they sleep until woken up and then look for work to do. Since all the slot I/O requests are independant of one another, the disk slaves can therefore all work independantly, permitting concurrent I/O on a node.

The ideal number of disk slave on a given node should be enough to theoretically always keep all of the local storage units busy. Since a disk slave is always either accessing a disk or working with the shared memory, then about 2 disk slaves per storage unit should do the job. An I/O node with 4 storage units (disks) should therefore have about 8 disk slaves running. Assuming even distribution of the I/O, then at any given time a storage unit will always have 2 disk slaves focused on it, keeping it potentially busy at all times.

### \*\* 6.7 Use of "physical pointers"

This performance boost actually doesn't effect the ability to perform overlapped I/O. However it should raise the efficiency of the persistent storage API.

A physical pointer allows the user to amortize the cost of the persistent pointer dereference over multiple accesses into persistent storage. This should lower the overall pointer dereference overhead, increasing the scanning speed of the query. This should have its greatest effect however when the persistent data is being populated; in that case the number of dereferences per object is significantly higher than during a

query.

## \* 7.0 Caveat Emptor

This system is not a panacea, and it has some notable limitations specifically having to do with the persistent object API. Known problems are listed in the following sections.

### \*\* 7.1 Non-portable object data

As discussed in section 3.3.3, stores really are not portable. Object data created on one system can't be presumed readable on another:

- 1) Padding within structures is compiler-dependant.
- 2) Byte-ordering is CPU architecture-specific.
- 3) Segment size is implementation-dependant.

This is a serious limitation if this system were to be considered as a new data interchange format. It just isn't designed with that in mind.

However, it is conceivable that a specific interchange format could be created, based upon some of the concepts of the object-structuring in persistent object managers. The API would have to be expanded such that it would be possible to "flatten" an address space or store of objects into a stream, and "expand" that stream back into a properly formatted address space or store at the destination machine. The concept here is simple: Use one (native) format for representing the data structure on the machine. This would be optimized for the speed and efficiency during queries. Use another (generic) format for representing the data structure in a compiler, CPU architecture, and implementation-independent way. This secondary format is optimized for compatibility, but not speed.

### \*\* 7.2 Major problems with member functions in persistent memory

Are persistent objects in this system really "objects"? No. They are really persistent structures. Why? Because the difference between a "structure" and an "object" is that an "object" is essentially a "structure" with protection and member functions. And it is NOT safe to use member functions with the persistent structures supported by [POPM] or for that matter by the UIC ptool implementation. Since member functions are not safe, protection is useless. That just leaves the "structure" part of the definition.

The following sections illustrate why it is generally not safe to use member functions in this system.

What happens if a member function were defined and called for a persistent object? It won't work properly. With normal member functions it can be made to work with a little bit of kludginess. With constructors, the work-around is more painful. But with virtual functions, there is no work-around.

Let's study the implications of the use of member functions in persistent objects:

#### \*\*\* 7.2.1 Nonvirtual member functions for persistent objects

First let us consider the case of a simple nonvirtual member function:

```
class abc {
public:
    void myfunction(void);
    int foobar;
};

typedef pptr<abc> ppabc;

void asubroutine(ppabc ap)
{
    :
    :
    ap->myfunction();
    :
    :
}
```

So what happens at the "ap->myfunction()" call? Well, the overloaded "->" operator gets called, which returns an appropriate pointer into a slot for the object in question. The compiler then uses this value as the "this" pointer in a call to abc::myfunction().

Interesting. So abc::myfunction() gets as its context a direct pointer into the slot. What happens if abc::myfunction() does further operations to manipulate persistent storage? If it does enough such work, the slot holding its own context may be reclaimed for use by another segment. If the member function then tries to access its class context, then it is either going to (a) read garbage, or (b) overwrite some other segment!

This problem occurs because the slot containing the object's segment must remain valid as long as there is an object pointer aimed at it. And in the case of a member function, that pointer remains aimed at it for the ENTIRE SCOPE OF THE MEMBER FUNCTION. This is a very large window compared to the simpler case of just accessing persistent member data within an expression. The pointer lifetime issue rears its ugly head again.

One really bad aspect of this situation is that the problem won't actually surface unless the slot in question gets recycled during the member function's execution. And that action depends on factors outside the scope of the executing process. For example, on a node with only one query running, that query process will likely get all of the available slots, making the above scenario relatively unlikely. But if multiple queries are running, the slots are going to be divided among the queries, making this problem far more likely. In other words, code debugged and certified in a test system may fail in a production environment because the outside conditions are going to be different. This is an exceptionally bad property of this problem.

To get around this problem, a physical pointer (as described in section 4.4.3) must be used to make the member function call. The physical pointer will hold a lock for that object and guarantee that the slot will not be reclaimed during the execution of that member function. It is unsafe to use normal persistent pointers when calling member functions. One very important caveat of work-around is that now every called member function has the potential to be locking a slot. If too many such calls are active, there is a possibility of deadlock. Yuck.

### \*\*\* 7.2.2 Constructors for persistent objects

It gets worse. Consider now the impact of constructors. A constructor function is just like a normal member function, in that it gets a "this" pointer and may internally perform other operations. So it is just as vulnerable to the pointer lifetime problem as normal member functions. Except now we have no way to bracket the constructor with a physical pointer! Why? Because before the constructor is called, there simply is no pointer to lock - the persistent pointer gets created as part of the "new" operation that calls the constructor. Locking it after the "new" statement is too late. The overloaded "new" operator can't do it either because the constructor is actually called after the "new" operator returns (but before the creating subroutine gets control back).

To work around this problem requires an even more restrictive covenant: The constructor body must not do any persistent operations. This is really hard to live with because it is very likely that the persistent object may contain pointers to other persistent objects which logically need to be created in the constructor. Or the constructor may need to dereference other persistent objects in order to fetch information for deposit into its instance. Cases like this can only be addressed by performing them outside of the constructor. Perhaps they can be done by a normal member function whose invocation has been protected by a physical pointer lock. \*sigh\*

### \*\*\* 7.2.3 Virtual functions don't work with persistent objects

Virtual functions are implemented in C++ by allocating a "hidden" virtual table pointer in the class structure. When such a function is called, the compiler generates a dereference of that field to find the virtual table. The virtual table is then indexed by the virtual function number to find the address of the actual function to be called.

This technique does not work in ptool, nor will it work in our query system. Why? Because the virtual table is not persistent data; it is logically part of the code for the program. So the virtual table pointer, in the persistent object, is pointing at an executable-specific table in the code. Worse still, the virtual function number, which is used as the index into the table, is executable-specific.

The upshot of all this is that if a persistent address space is populated and queried with the same executable program, then virtual functions in persistent objects would be usable. However, if the executable were relinked, or different programs were used to access the persistent address space, then the first call to a virtual function will dereference random code memory and either result in a segmentation fault (if the dereference failed), or an instruction trap (if it got as far as calling through the bogus function pointer in the bogus virtual table).

Therefore it is not possible to employ virtual functions in persistent objects, as implemented in this system. If a user tries to write code in this manner, this violation will also be impossible to detect until the code executes (since the code will compile ok).

Local Variables:  
mode:indented-text  
fill-column:75  
eval:(outline-minor-mode 1)  
End: